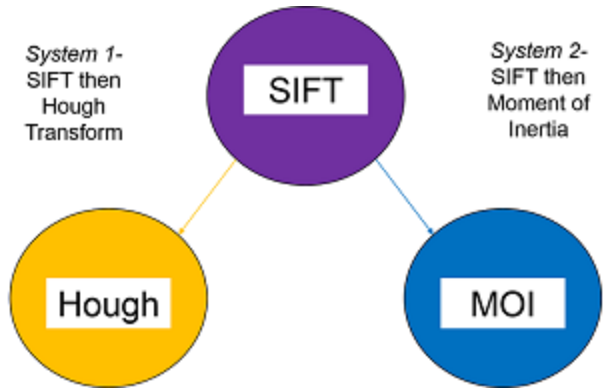
Background

Introduction

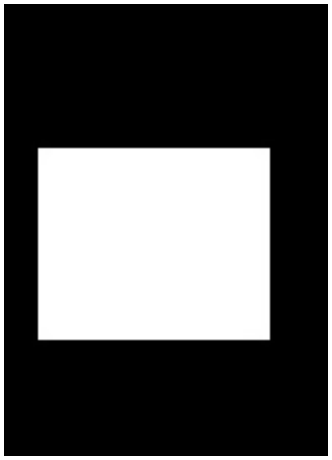
Object recognition is one of the hottest areas in computer vision. Finding and identifying objects in an image is fundamental to facial recognition, object tracking in videos, object matching and many other applications. To achieve object recognition (from images), one first must be able to extract distinct features of the image, then correctly identify the object of interest from these key features. Our group explored three different algorithms. The first, Scale-Invariant Feature Transform, was a method to obtain the distinct features of the image while the other two, Hough Transform and Moment of Inertia, were pattern recognizing algorithms.

Motivation

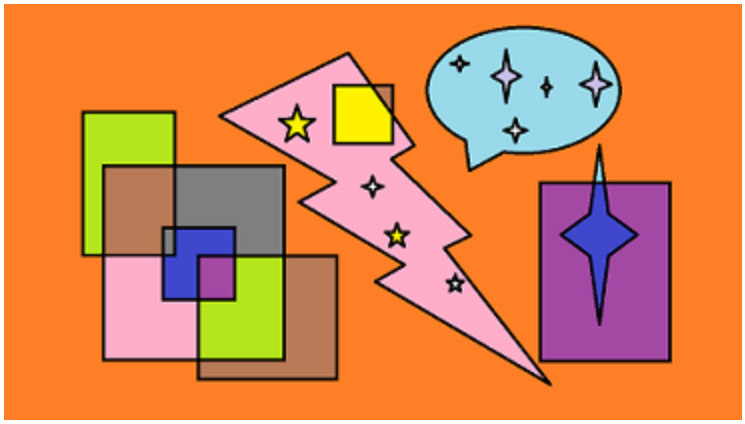
We wanted to simulate a user being given a sheet of paper with squares and other shapes on it. The user would tell us how many squares appeared in the image. Because we were highly interested in the field of object recognition we decided to implement two systems and compare the results. Both systems used Scale-Invariant Feature Transform to obtain feature points. The first system then passed those results into the Hough Transform. The second system used the feature points to implement a formula involving the properties of the Moment of Inertia.



Flow of the system design.



Our square template.



An example test image.

Implementation of Scale-Invariant Feature Transform

Introduction

The Scale- Invariant Feature Transform is an algorithm in computer vision to detect and describe points of interest in an image.

Approach

According to Prof. Dowe's paper on Distinctive Image Features from Scale-Invariant Keypoints(2004), there are four stages to SIFT:

1. "Scale-space extrema detection"- Searches the entire image for candidate interest points
2. "Keypoint localization"- Calculate the location and scale for each candidate, remove candidates that are not stable
3. "Orientation assignment"- Assign each key point with one or more orientations that are calculated based on the gradient direction at that key point location in the image
4. "Keypoint descriptor"- For each key point, calculate the gradient in its surrounding area. This allows the transform to be distortion resistant. The above approach "transforms image data into scale-invariant coordinates relative to local features".

Our project seeks to achieve scale, rotation and translation resistance. However due to time-constraint, we did not implement stage 4 "Keypoint descriptor" of SIFT.

Implementation

Stage 1

Apply Gaussian filters of different scales to the image. By using different scales the Gaussian filters would have different variances. Due to the inherent properties of Gaussian filters, this would "smooth" out the images, removing finer details of the image. At different scales, the details of the

image that are insignificant compared to the standard deviation of the Gaussian filter applied would be removed. The Gaussians are generated using the following formula:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

Then the image, represented as an array of digits, is convolved with the Gaussian.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

L(x,y,sigma) is the value of the resulting image at location (x,y) under the Gaussian filter with standard deviation sigma. I stands for the original image.

We applied Gaussians with scale 0, 1, and 2 to the image. At scale 0, we are essentially preserving the original image, at scales 1 and 2 we are "smoothing out" the image to an increasing extend. We have 3 octaves of resulting images, each octave consists of images resulting from repeated applying the gaussian filter of the same scale to the original image. After each octave, the image is down-sampled by two.

Code

```
%1st Octave
Octave1 = [];
ki=0;
Image(x:x+4,y:y+4)=0; %zero-pad
for kj=0:3
    kk=sqrt(2);
    sigma=(kk^(kj+(2*ki)))*1.6;
    %generating the gaussians
    for m=-2:2
        for n=-2:2
            gaussian1(m+3,n+3)= (1/((2*pi)*((kk*sigma)*(kk*sigma))))*exp(-((m*m)+(n*n))/(2*(kk*kk)*(sigma*sigma)));
        end
    end
    %convolve image with the gaussian filters generated above
    for i=1:x
        for j=1:y
            singlesum=Image(i:i+4,j:j+4)'.*gaussian1;
            conv1(i,j)=sum(sum(singlesum));
        end
    end

    Octave1=[Octave1 conv1];
end
```

Stage 2

Now we have the image smoothed to different extends, with variant amount of fine detailed preserved in the resulting images. Within each octave, we use Difference of Gaussian, which is basically subtracting neighboring images from each other. Difference of Gaussian is proven to be a close approximation of scale-normalized Laplacian of Gaussian, which is shown to "produce the most stable image features compared to a range of other possible image functions, such as the gradient, Hessian, or Harris corner function". Moreover, Difference of Gaussian is efficient to compute since it's just subtracting images.

Code

```
%%Difference of Gaussian
% differnce of gaussian for octave1
diff_11=Octave1(1:512,1:512)-Octave1(1:512,513:1024);
diff_12=Octave1(1:512,513:1024)-Octave1(1:512,1025:1536);
diff_13=Octave1(1:512,1025:1536)-Octave1(1:512,1537:2048);
 % difference of gaussian for octave2
diff_21=Octave2(1:256,1:256)-Octave2(1:256,257:512);
diff_22=Octave2(1:256,257:512)-Octave2(1:256,513:768);
diff_23=Octave2(1:256,513:768)-Octave2(1:256,769:1024);
% difference of gaussian for octave3
diff_31=Octave3(1:128,1:128)-Octave3(1:128,129:256);
diff_32=Octave3(1:128,129:256)-Octave3(1:128,257:384);
diff_33=Octave3(1:128,257:384)-Octave3(1:128,385:512);
```

Then for each pixel in a resulting image, we compare it to its eight neighboring pixels in the same image and nine neighboring pixels in the images processed by adjacent scales. It’s selected if it’s greater or smaller than all its neighbors. The result is a candidate key point.

Code

```
%for maximum
%compare the pixel with it's neighbors in the same image
if (((diff_12(i,j)>diff_12(i-1,j))&&(diff_12(i,j)>diff_12(i+1,j)))....
    &&(diff_12(i,j)>diff_12(i,j-1))&&(diff_12(i,j)>diff_12(i+1,j+1)))....
    &&(diff_12(i,j)>diff_12(i-1,j-1))&&(diff_12(i,j)>diff_12(i-1,j+1)))....
    &&(diff_12(i,j)>diff_12(i+1,j-1))&&(diff_12(i,j)>diff_12(i,j+1))))
        x1=x1+1;
else
    continue;
end
if x1>0
    %compare the pixel with its neighbors in the image processed by gaussian of
    %1 scale more
    if((diff_12(i,j)>diff_13(i,j))&&(diff_12(i,j)>diff_13(i-1,j)))....
        &&(diff_12(i,j)>diff_13(i+1,j))&&(diff_12(i,j)>diff_13(i,j-1)))....
        &&(diff_12(i,j)>diff_13(i+1,j+1))&&(diff_12(i,j)>diff_13(i-1,j-1)))....
        &&(diff_12(i,j)>diff_13(i-1,j+1))&&(diff_12(i,j)>diff_13(i+1,j-1))&&(diff_12(i,j)>diff_13(i,j+1)))
        y1=y1+1;
    else
        continue;

    end
end
%compare the pixel with its neighbors in the image processed by gaussian of
%1 scale less
if y1>0
    if ((diff_12(i,j)>diff_11(i,j))&&(diff_12(i,j)>diff_11(i-1,j)))....
        &&(diff_12(i,j)>diff_11(i+1,j))&&(diff_12(i,j)>diff_11(i,j-1)))....
        &&(diff_12(i,j)>diff_11(i+1,j+1))&&(diff_12(i,j)>diff_11(i-1,j-1)))....
        &&(diff_12(i,j)>diff_11(i-1,j+1))&&(diff_12(i,j)>diff_11(i+1,j-1))&&(diff_12(i,j)>diff_11(i,j+1)))
            z1=z1+1;
    else
        continue;
    end
end
```

Stage 3

To calculate the magnitude and orientation of each key point, we look at all it’s neighboring pixels in the image that is processed with the same scale.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

m(x,y) stands for the gradient magnitude of the point and theta(x,y) stands for the orientation of the point.

Code

```
%Calculating Magnitude and Orientation
for i=2:511
    for j=2:511
        %calculating magnitude and orientation of each point by looking at
        %its neighbors
        magsquared(i,j)=((diff_12(i+1,j)-diff_12(i-1,j))^2)+((diff_12(i,j+1)-diff_12(i,j-1))^2);
        orientation(i,j)=atan2((diff_12(i,j+1)-diff_11(i,j-1)),(diff_12(i+1,j)-diff_11(i-1,j)));
    end
end
mag=sqrt(magsquared);
```

Implementation of Hough Transform

Introduction

The Hough Transform is a feature extraction technique that estimates parameters of a shape from its boundary points. Advantages of the Hough Transform include that it is scale-invariant, shift-invariant and rotation-invariant. It also functions well with added noise. Disadvantages include that it is computationally expensive and that it can falsely detect multiple instances of a single edge.

Approach

The Hough Transform converts points in an edge picture to sinusoids in a parameter space expressed in polar coordinates.

These sinusoids are put into an accumulator. When two points are collinear, the sinusoids corresponding to the two points intersect at a point. These intersections form peaks in the Hough domain. The peaks in the accumulator tell us which features in an image are present. For example, a square has four Hough peaks, corresponding to each corner in a square.

Features that the Hough Transform can detect include lines, points, and curves such as circles and ellipses in images because these features have parametric representations.

Implementation

1. Convert Image to binary scale

a.The first step we need to do is to convert color images to binary images such that the only region of interest are the boundaries of shapes and not the varying colors the shape may have.

```
Template = imread('Template.jpg');

Image = imread('TestImage2.png');

bwTemplate = im2bw(Template);
bwImage = im2bw(Image,0.1);
```

2. Given the binary image, we use the Matlab edge function to detect the edges

```
% [gTemplate ,t ] = edge(bwTemplate,'Canny');
% [gImage ,t ] = edge(bwImage,'Canny');
```

3. From the edges we compute the Hough Transform

```
[H,T,R] = hough(BW,'RhoResolution',0.5,'Theta',-90:0.5:89.5);
```

4. Finally, we calculate the Hough peaks

```
% detect Hough peaks
numpeaks = 10; %Number of peaks to look for
P = houghpeaks(H, numpeaks);
```

Implementation of Moment of Inertia

Introduction

To achieve scale, translation and rotation invariance in object recognition, we need to identify intrinsic traits of the object that are not affected by these operations. Moment of Inertia is mass property of a rigid body that determines the torque needed for a desired angular acceleration about an axis of rotation. Moment of inertia depends on the shape of the body and thus can be used as parameter for simple object recognition.

Approach

Moment of Inertia can be calculated if given centroid of the object and it's mass.

$$I_P = \sum_{i=1}^{N} m_i r_i^2.$$

I is Moment of Inertia, m is the unit mass at a location of distance r from the centroid, and r is the distance from the centroid.

However a two-dimensional image does not have the mass to be used for calculation of Moment of Inertia. We can instead use the intensity of each pixel as mass of a particle located at the pixel. With that, we can calculate the Moment of Inertia of our images. Since we are only concerned with the shape of the object, we can convert the images to black and white (with intensity of each pixel being either 0 or 1) and consequently the intensity of each pixel can only take on 2 values and the Moment of Inertia can be much more easily calculated given this simple duality.

Actual Formula used-

$$I = \sum_{i=1}^{N} r_i^2 = \sum_{i=1}^{N} ((x_i - C_x)^2 + (y_i - C_y)^2)$$

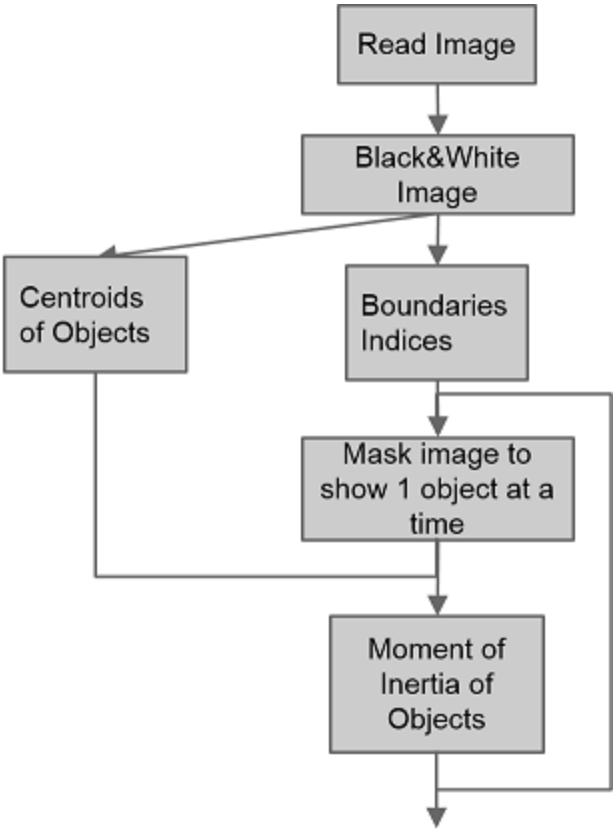
x-coordinate with pixel value of 1

x-coordinate of centroid

y-coordinate with pixel value of 1

y-coordinate of centroid

Diagram of Process Used to Calculate Moment of Inertia-



Code

Main Functions Used in Matlab:

- im2bw- change image to black and white

- bwboundaries- getting boundary indices of object
- regionprops- getting centroids

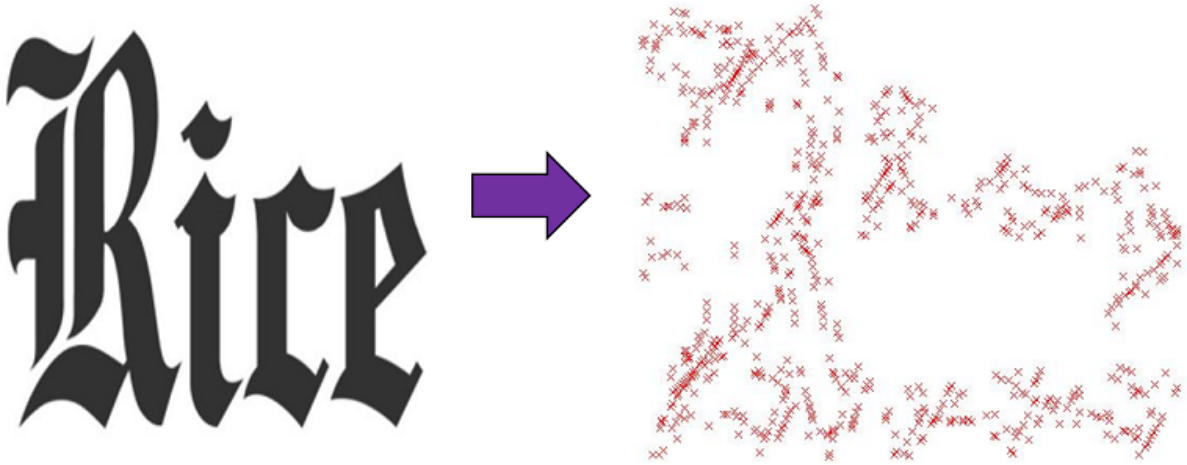
```
function nMomentOfInertia=MomentofInertia(binaryimage,centroid)

    length=size(binaryimage,1);
    width=size(binaryimage,2);
    MomentOfInertia=0;%MOI

    for i=1:length
        for j=1:width
            if binaryimage(i,j)
                MomentOfInertia=((i-centroid(1))^2+(j-centroid(2))^2)+MomentOfInertia;
            end
        end
    end

    NumberOfPixels=sum(sum(binaryimage));
    nMomentOfInertia=MomentOfInertia/(NumberOfPixels.^2);
    %normalized I with total number of pixels

return
```

```
function  maskedimage=masking(binaryimage,NumberofObjects)
boundaries = bwboundaries(binaryimage); %boundary indices
length=size(binaryimage,1);%size
width=size(binaryimage,2);

for k=1:NumberofObjects
    maskedimage{k}=binaryimage;
    for i=1:length
        for j=1:width
            in = inpolygon(i,j,boundaries{k}(:,1),boundaries{k}(:,2));
            %detects if a point of interest is within the boundary
            %then mask all points within boundary to be 1 and everywhere
            %else to be 0.
                if (in)
                    maskedimage{k}(i,j)=1;
                else
                    maskedimage{k}(i,j)=0;
                end
        end
    end
end

return
```

Results

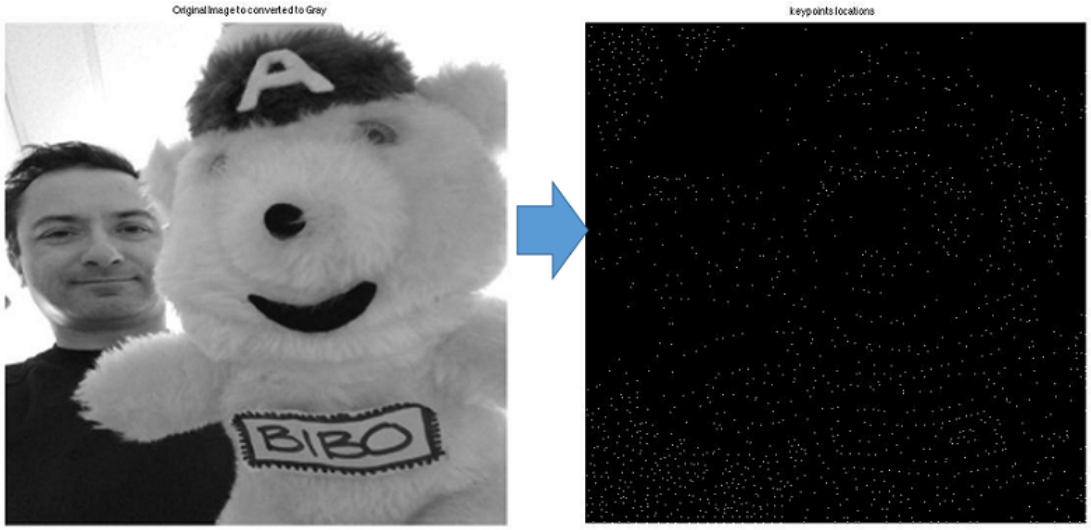
Scale-Invariant Feature Results

SIFT Results on image of Rice



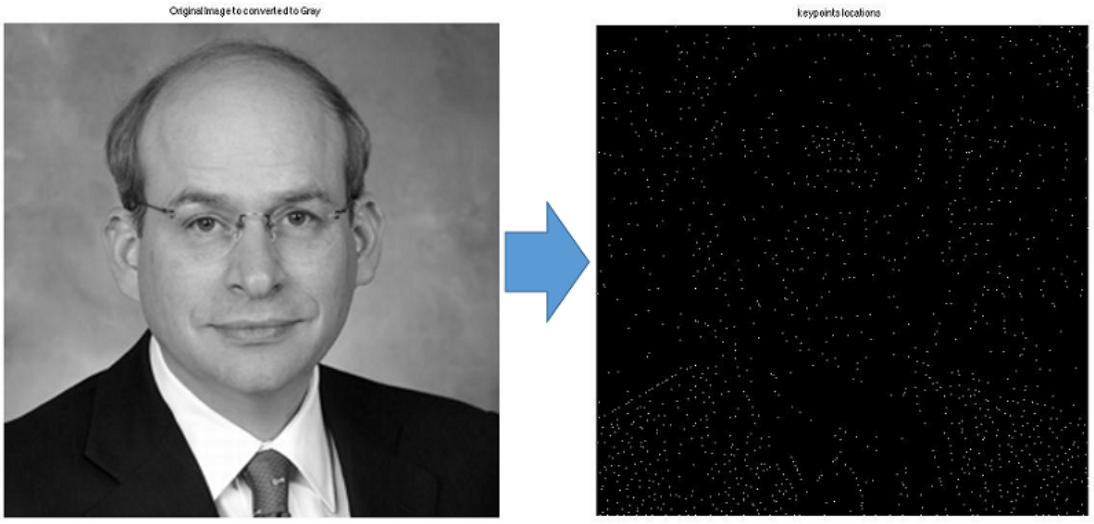
The curves are well detected but horizontal and vertical lines are lost.

SIFT Results on BIBO Bear



The words BIBO can be seen on the bear's chest.

SIFT results on President Leebron

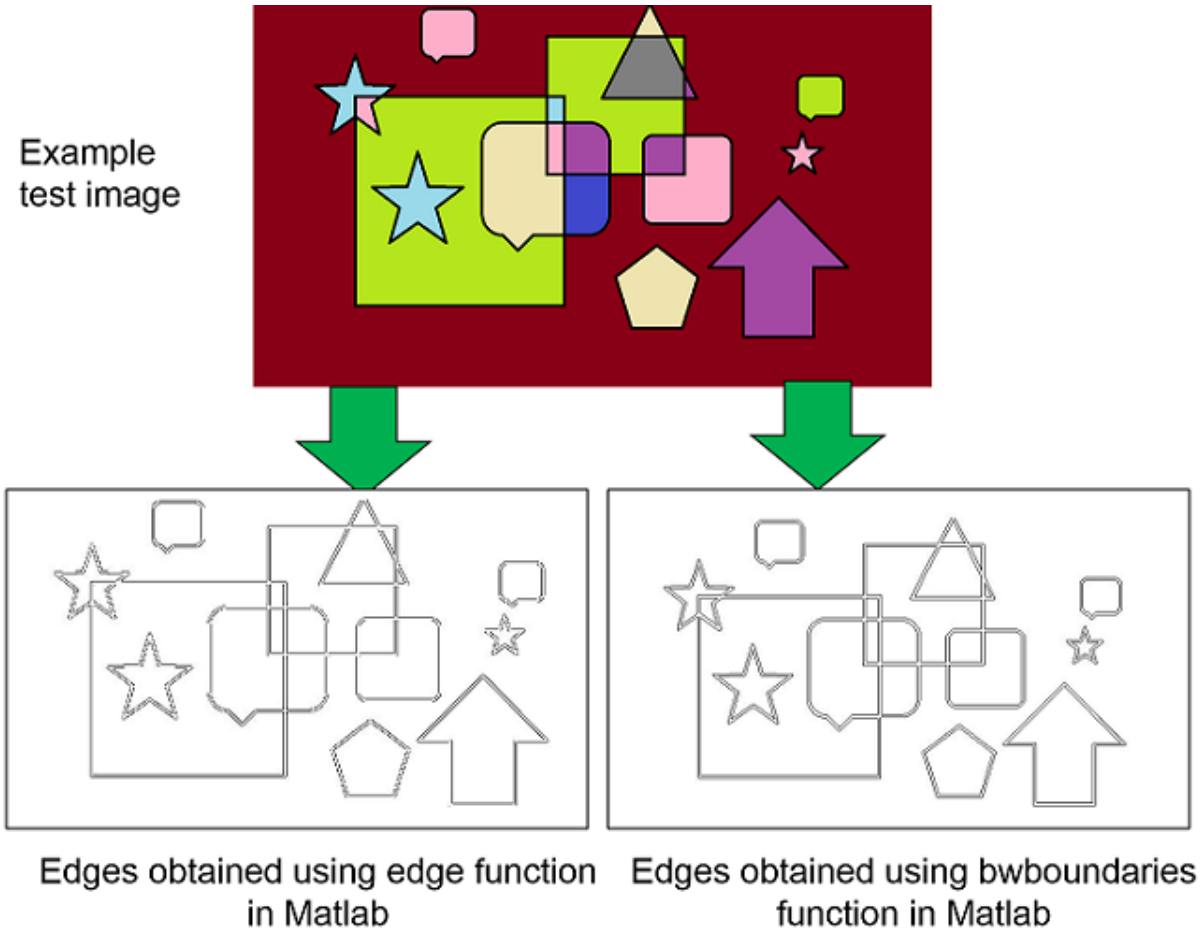


It is evident that it is the image of a person.

Although SIFT is extremely useful, it cannot easily find vertical or horizontal lines because they are not considered feature points. Since our image templates were made up of almost all straight lines, we had a significant issue. Due to time constraints we chose to implement the edge functions in MatLab. Instead of passing the results of SIFT into the Hough Transform and Moment of Inertia we passed the results of the edge functions in MatLab.

Results of Matlab Functions

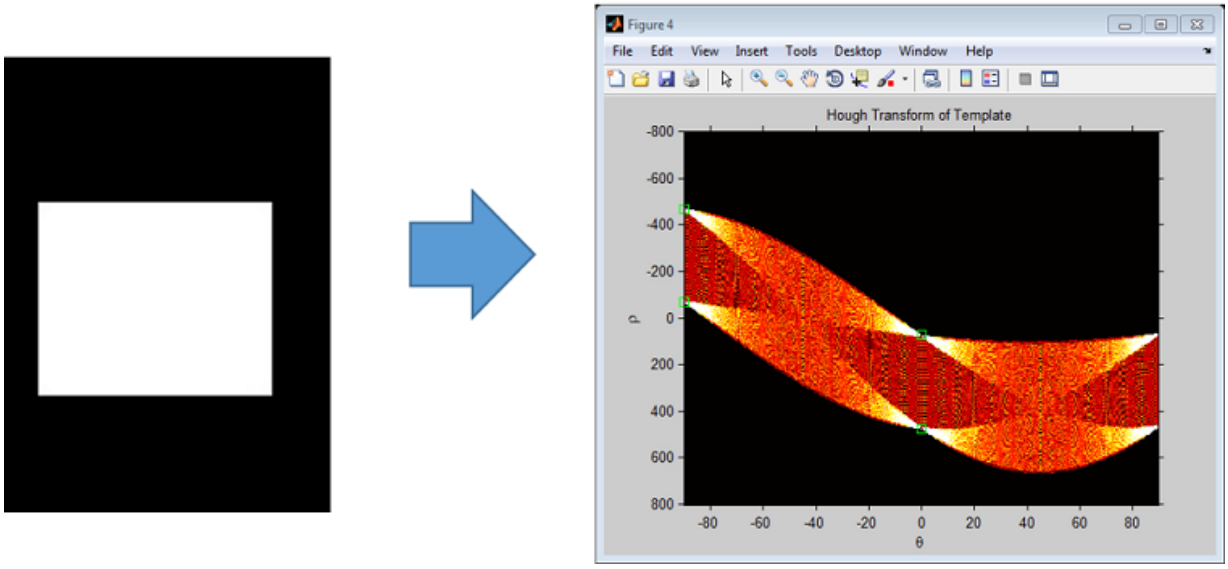
Results of Passing a Test Image into Matlab Edge Functions



Clearly the edge results from matlab are much more suited to our type of test images.

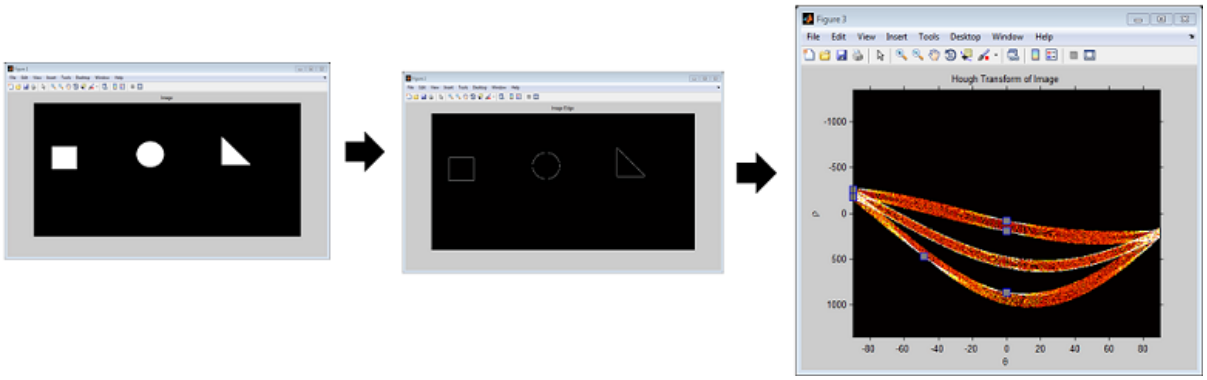
Hough Transform Results

Hough Transform of Square Template with Hough Peaks in Green

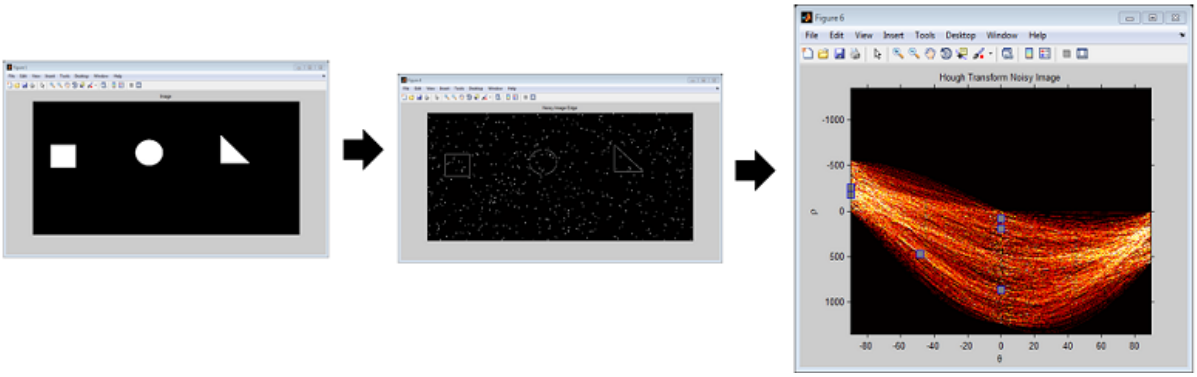


The Hough Transform of the template image.

Hough Transform of Test Image 1 with Hough Peaks in Blue

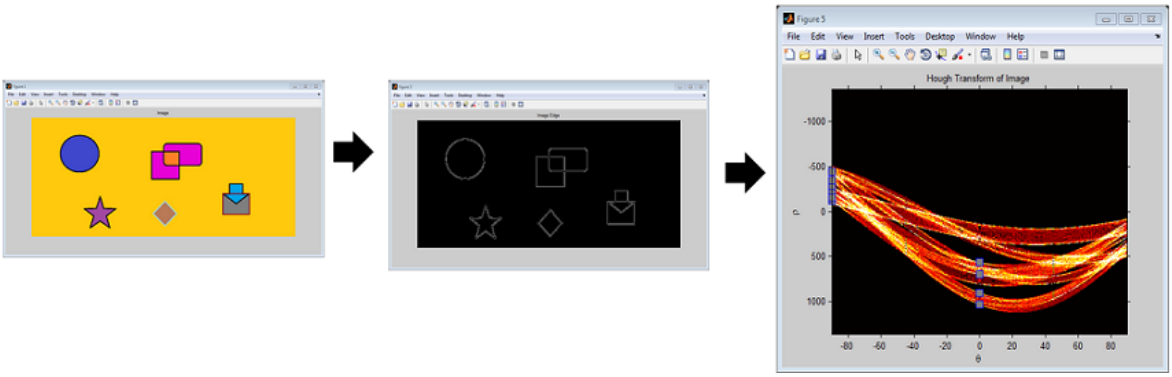


Hough Transform of Test Image 1 (0.1% Salt and Pepper Noise) with Hough Peaks in Blue

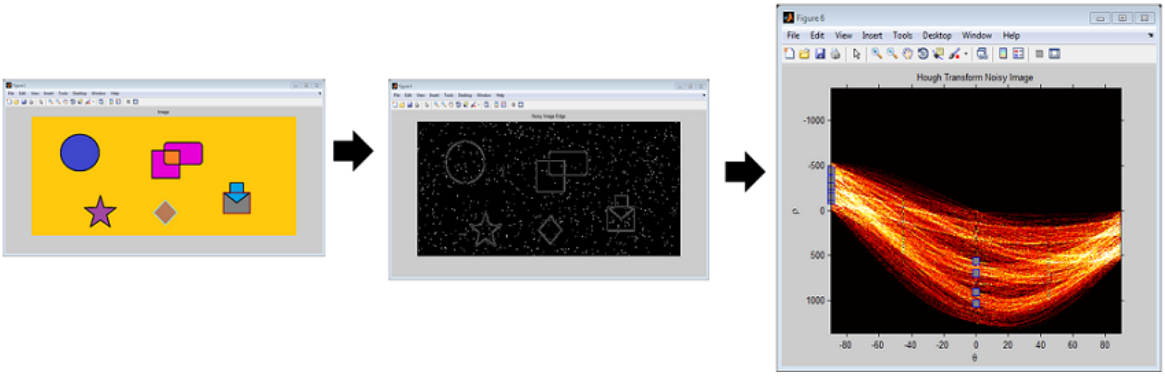


Even with noise same peaks are found.

Hough Transform of Test Image 2 with Hough Peaks in Blue

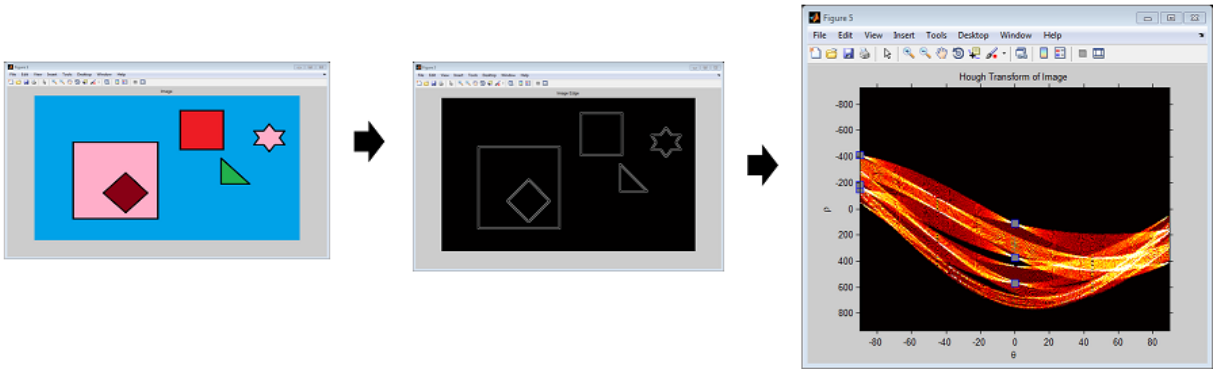


Hough Transform of Test Image 2 (0.1% Salt and Pepper Noise) with Hough Peaks in Blue

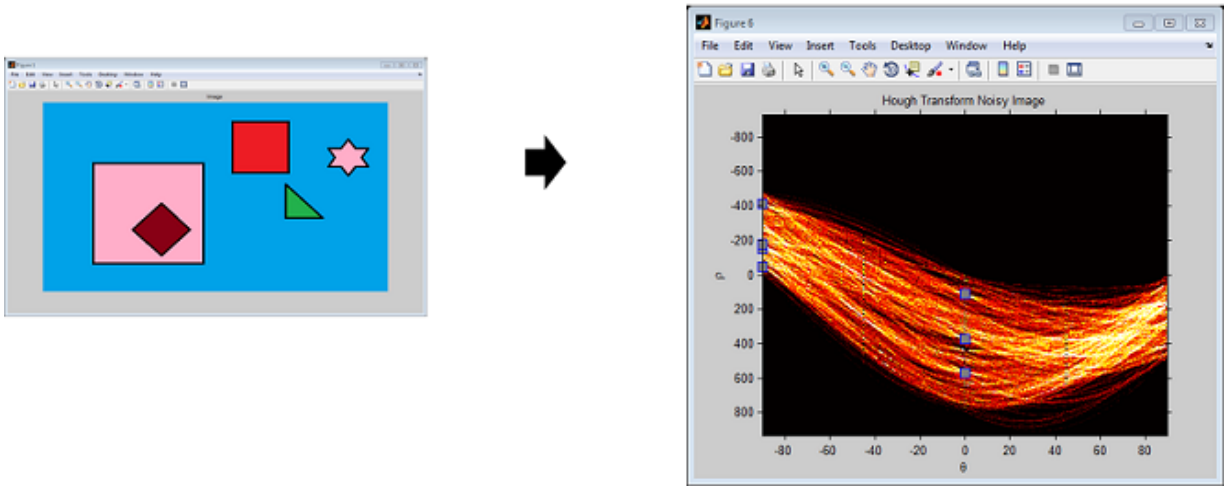


Same peaks are found with noise.

Hough Transform of Test Image 3 with Hough Peaks in Blue



Hough Transform of Test Image 3 (Gaussian Noise mean 0 variance 0.0001) with Hough Peaks in Blue



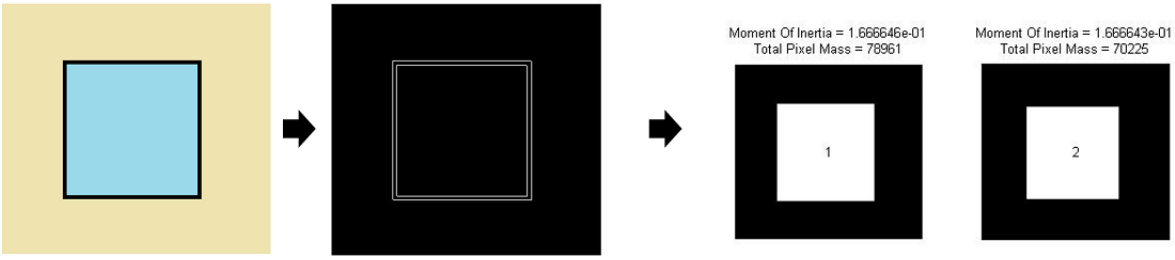
With this amount of noise the peaks are still found. As long as the noise does not interfere with the edges the peaks will always be found. However, if the edges are lost then the Hough Peaks will be incorrect.

Moment of Inertia Results

Description of Steps in Demo

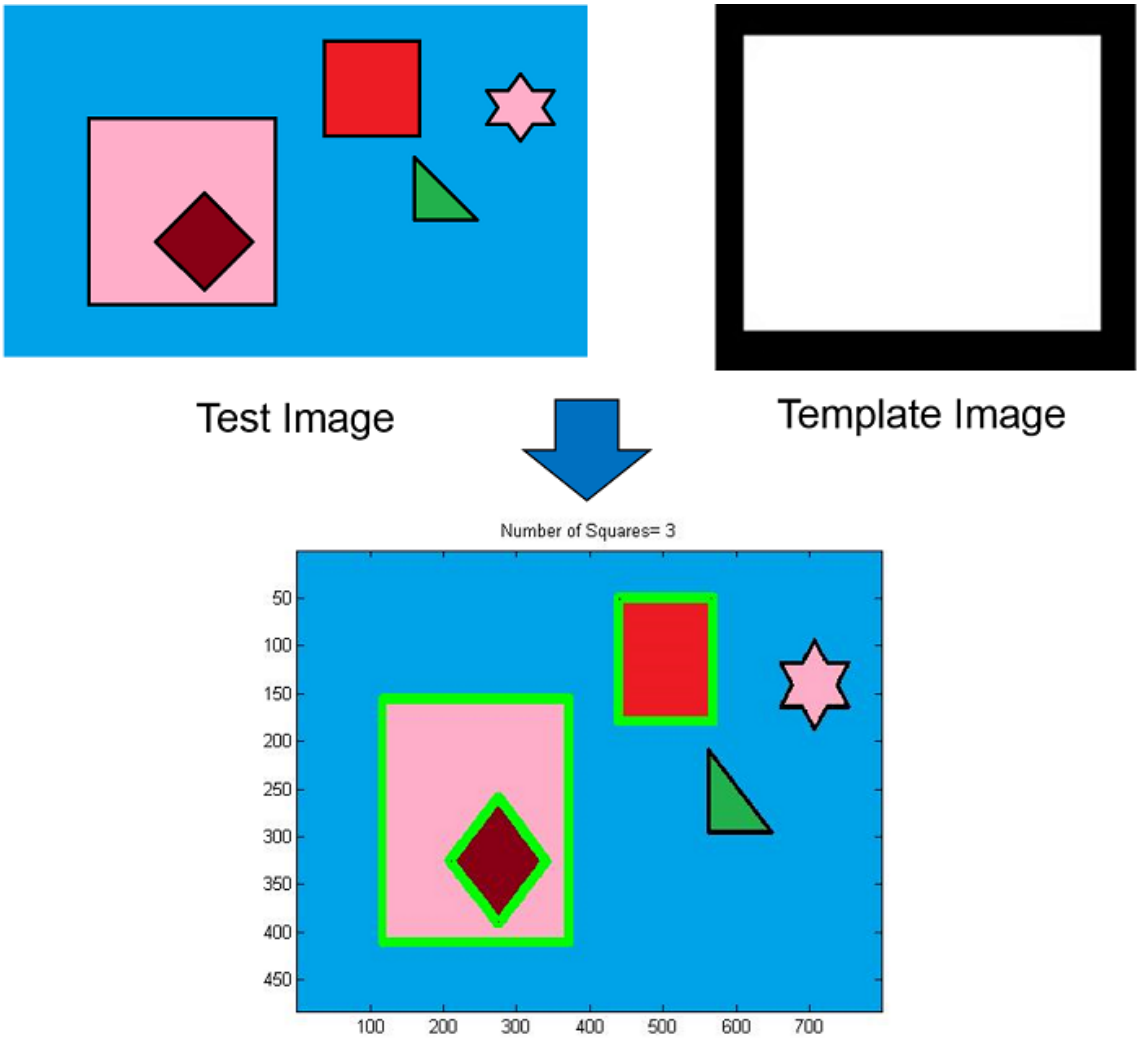
1. Read in the image as a matrix of type ‘double’ and turn it into a logical array (im2bw function)
2. Using bwboundaries functions, find all connected boundaries (default is 4-connectedness of pixel)
3. Iterate through all closed boundaries and mask all pixels within boundary to be 1 and 0 outside the boundary
4. Find the moment of inertia of the masked images which is normalized by the square total number of pixels

A demo to show how Moment of Inertia works



There are two boundaries per shape because the outline of the image is a black line of a certain width. Although the two masked shapes resulting from this duplication have different Total Pixel Mass, they have almost identical normalized Moment Of Inertia. Hence in the implementation of this algorithm, the total number of objects recognized is halved to eliminate duplicates.

Moment of Inertia Method on Simple Image

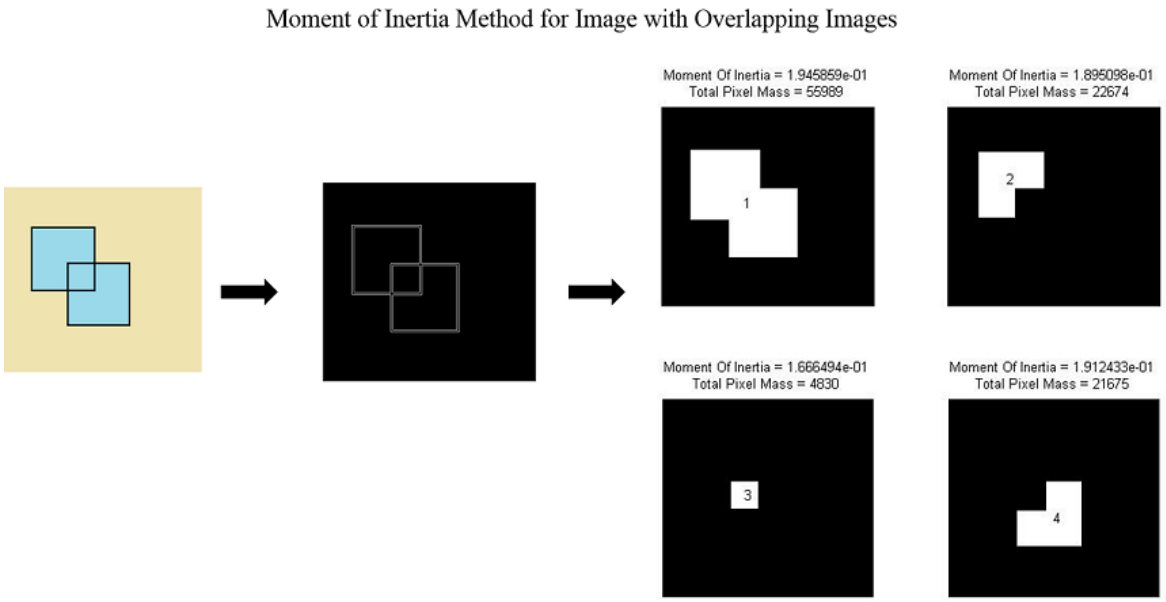


Moment of Inertia method on a simple image without noise. 3 squares are correctly identified.

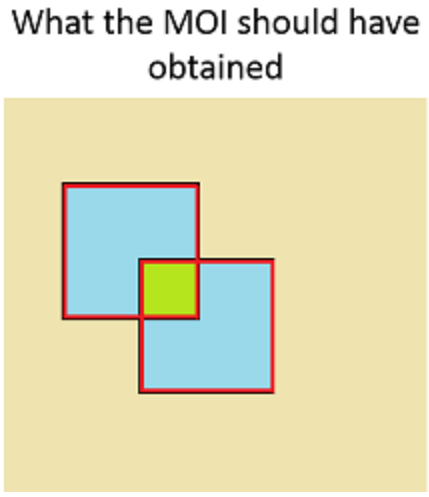
Moment of Inertia Method works perfectly on a test image without overlaps where the shapes are significantly different.

However, bwboundaries returns closed boundaries which would create problems when you have intersection of lines.

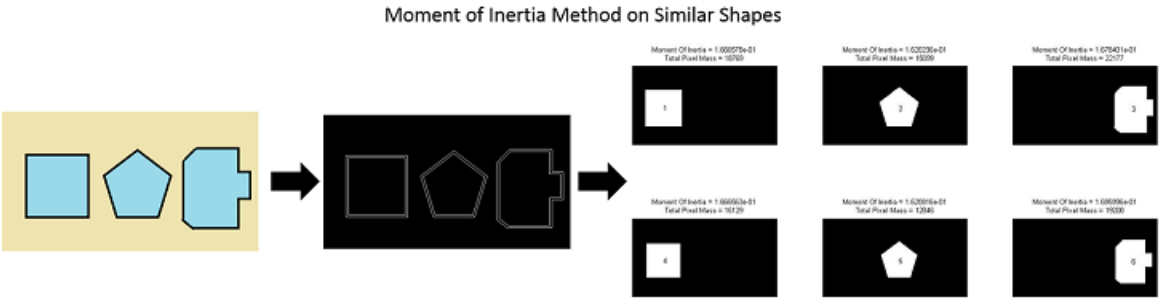
Moment of Inertia Method on Image with Overlaps



We can clearly see from the previous images that the function now detects 4 objects instead of the three that we are interested in, shown below as the two larger squares traced out by red outline and the smaller one filled by the color green.



Moment of Inertia Method on Image with Similar Shapes



In the above image, objects in the top right and top left corners of the subplots differ in moment of inertia by 0.703%, in spite of having drastically different shapes. However, we note that such algorithm would only produce false positive.

Conclusion and Future Work

Conclusion

Scale-Invariant Feature Transform

After performing SIFT on a number of images and finding the location of the key points on those images, we realized that although SIFT recognizes the sharp corners and curvatures very well, it would not recognize key point along straight lines. Looking back at the SIFT algorithm, we found out that this is because straight lines are not scale and rotational invariant as compared to corners. Since SIFT focuses on the robustness of key points and on the features of an image, straight lines do not qualify as features.

Since, our goal is to identify squares in an image mainly comprised of geometric figures we are mostly dealing with straight lines. Hence, SIFT would not be a suitable algorithm for us to locate the edges in the image. We would need to resort to other methods. Due to time constraints we turned to Matlab edge Functions.

Hough Transform

The Hough Transform correctly identified the squares in an image but was computationally expensive. Every single pixel in the image had to be looked at. Also if enough noise was added that an edge was lost the Hough transform ceased to work at all. Overall the Hough Transform is a more robust approach because it can be generalized to even more complex situations where the Moment of Inertia method cannot.

Moment of Inertia

The Moment of Inertia method proved to be extremely rudimentary. It worked on simple images where the shapes had very different Moments of Inertia and the shapes did not overlap. But if the shapes overlapped the

boundaries were lost and the shapes were not identified correctly. Also, if two shapes had similar Moments of Inertia the system would give a false positive. Obviously Moment of Inertia is not a method that can stand on its own. However, it only gives false positives. It won't accidentally mask out an actual result. Due to that and the fact that it is computationally very cheap, it could be used as a preprocessor for a more computationally expensive algorithm.

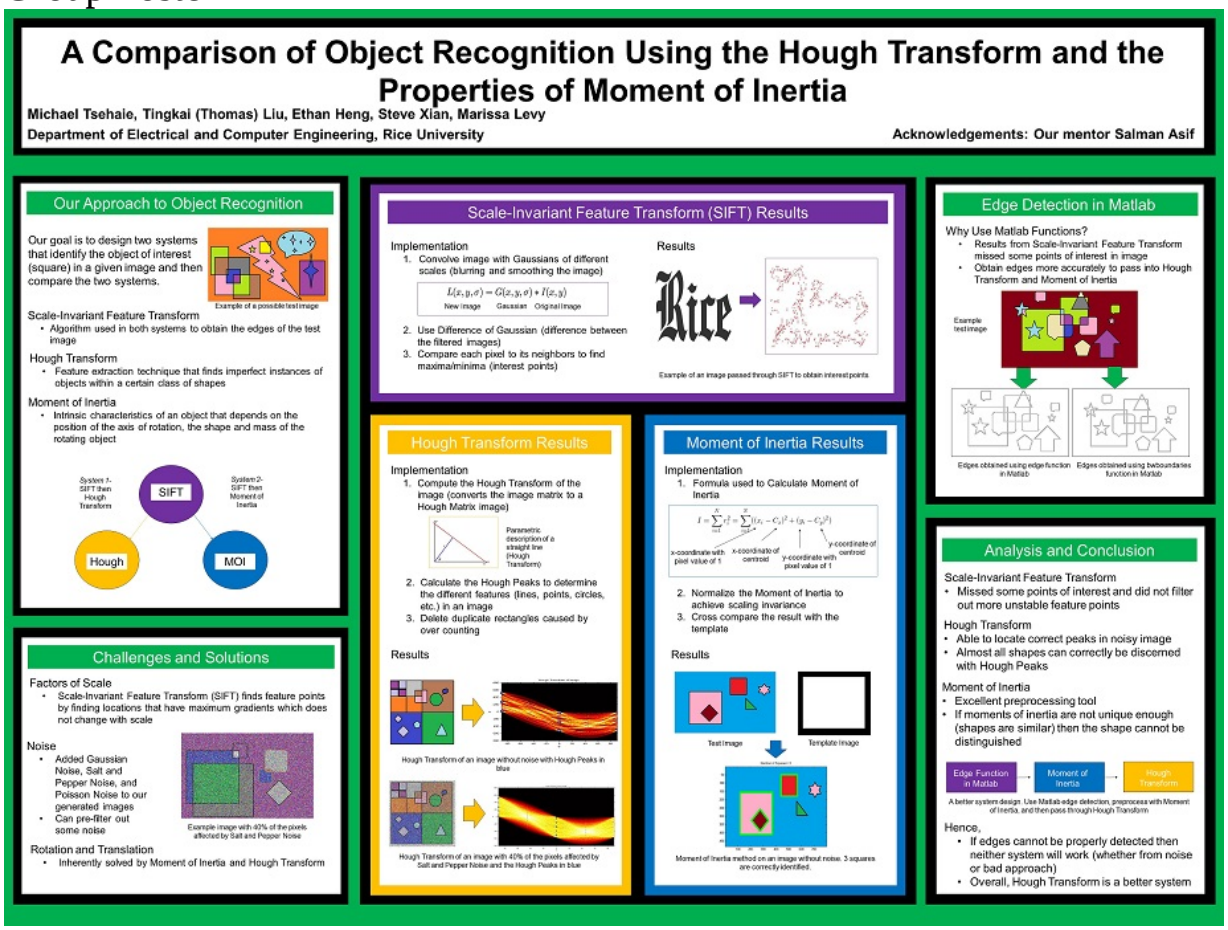
Future Works

Continuing on from here, we want to optimize our implementations. After successfully improving the speed of our code we would want to solve the issue our Moment of Inertia method has with overlapping images. This could be solved by starting from the smallest shape and moving outward. Finally, we would reorganize our system to match the following design:
New Design Model

A better system design. Use Matlab edge detection, preprocess with Moment of Inertia, and then pass through Hough Transform

Implementing this design would save computation time in the Hough transform. The Moment of Inertia would mask out shapes that were clearly not correct which would reduce the number of Hough Peaks and reduce the number of calculations. After implementing this improved system we would want to create our own way of obtaining the edges of our images besides Matlab.

Poster
Group Poster



Poster we used for presentation.